Winter 2019 - COMP 520 Compiler Design
Milestone 3 Report

Group 07
Zhuocheng Du - 260673297
Anqi Li - 260654876
Zisheng Wang - 260619653

1 Introduction

After milestone 2, the compiler should be able to perform lexical, syntactic and semantic analysis on input programs written in GoLite programming language. During milestone 3 and 4, code generation, which is converting an input program in GoLite to a semantically equivalent output program in another language, will be implemented. Our project will be generating code from GoLite to Python 3. Before diving deep into three areas of Go's semantics in the next section, we will first list a few known similarities and differences between Go and Python 3.

There are a few known advantages of Python. First, Python is dynamically typed, so we do not have to handle blank identifiers in any special way. Second, slice syntax, underline equality of objects, multiple assignments are all natively implemented in Python. In order to avoid possible variable naming conflict, we will add "___" (three underscores) as prefix to all variables, this avoids conflicts with all Python keywords. We will also have a temporary variable name generator function with the format of "___{int}" for any potential use in function parameters or anonymous struct.

A few notable differences between GoLite and Python:

- Python does not have arrays, so we will use the combination of "None" , "count()" and "list()" constructs in Python to simulate the behavior of statically sized arrays and dynamically sized arrays (slice's underline structure) of GoLite.
- Python only has function scoping, so we will wrap all block statements of GoLite in unparameterized functions.
- Python does not have switch statements. We will replace those with if-elif-else constructs. To handle break statements, we can simply use return, since all block statements are wrapped inside functions.
- Python does not have increment and decrement statements, so those will be transformed into compound assignment.
- Python's for-loop is very different from that in GoLite. We will replace GoLite's for-loop with while-loop in Python.
- Python does not have structs, but "dict()", combined with "deepcopy()", can be used to simulate their behavior.
- We will double-check GoLite's pass-by-value and pass-by-pointer behavior involving complex objects at a later stage. We will pay particular attention to slices and nested/inner slices.
- We will double-check print format at a later stage.
- The default float formatting in Go consists of 3 exponential digits, whereas Python's scientific float formatting only provide 2 exponential digits. Although their precision can be specified, the number of exponential digit cannot. If time allows, we will wrap the float64 of Go inside a custom class of Python to have their print format displayed correctly.
- Python does not have an explicit bitClear operator, therefore we will alter all op-assignment involving bitClear.

2 Semantics and Mapping Strategy

In this section, we describe the semantics of three of the key areas of Go programming language. We also discuss, for each of the chosen key areas, how to map it to Python 3.

2.1 Variable declarations

Variables of Go can only be declared once in each scope (with the blank identifier being an exception). Our typechecker shall ensure that the input program is valid, and since Python is dynamically typed, we do not have to implement anything special in this regard.
However, if a Go variable is not explicitly initialized, they are initialized to a default value. While there is a few exceptions in GoLang, such as individual element of a slice declared with make() or new(), those exceptions shall not be encountered when using GoLite.
The following is a list of initial values for each type supported by GoLite, and their corresponding mapping in Python in case it differs:

- Int: 0
- Float64: 0.0
- Boolean : False
- Rune: 0
- String: “”
- Slice: nil → list()
- Arrays and structs initialize each element to their default value
    - Array: a list but with internal values predefined eg. int array size 5, [0 for _ in range(5)]
    - Struct: a dictionary with key-value pairs initialized to their default value

Since Python does not require variables to be explicitly declared before assignment, we translate the declaration statement of GoLite to declaring and assigning those variables in Python with those default values.

2.2 If statements

If statements are used to control the execution of different branches based on the evaluation result of boolean expressions associated with the branches. In Go, a simple statement is allowed immediately after the “if” keyword, and it will be executed before the if condition is evaluated. An if branch gets executed if and only if the corresponding condition evaluates to true. Zero or more elif statements can follow an if statement. An elif branch gets executed if and only if every if/elif branch before it has a false condition and its own condition evaluates to true. An else statement can come immediately after an if statement or if/elif statements. The else block does not get executed unless every if/elif branch before it has a false condition.

If-elif-else statements in Go can be mapped to Python 3 using functions. Since each if/elif/else block opens a new scope in Go but block scopes do not exist in Python 3, we need to map each if-elif-else block to an unparameterized function in the output program. If there is a simple statement following the “if” keyword in the input program, when performing the mapping, we need to extract it and put the equivalent statement written in Python 3 as the first line inside the unparameterized function mapped to by the if statement. In the case where a simple statement follows an “else if” keyword, we will decompose this into an “else” statement and an “if” statement, then insert the simple statement into the else statement.

Moreover, when there are nested if/elif/else statements, they will be mapped to nested functions in Python 3. All those functions can be named “___”. Since there is no such a thing as “recursive if statement”, the overshadowing of “___” will not cause any issues. Note that we will be prefixing “___” to all names from

the input program, so there is no chance that this will cause a conflict with anything. Finally, all mapped functions will be called immediately after their definitions.

2.3 Loops

The for statements in GoLang are used to execute a block repeatedly. There are three types of for statements supported by GoLite: infinite, while, and 3-part style. We will start with the 3-part style for loop. This style of for loop consists of an init statement, a condition, and a post statement. Both the init statement and the post statement can be defined using a simple statement. The init statement may be a short variable declaration, while the post statement may not. Before the for-loop is iterated for the first time, the init statement will be executed. Then, in each iteration, Go will check whether the condition is satisfied. If it is satisfied, the body of the for loop will be executed. Once the execution of the body is finished, the post statement will be executed. Note that if the condition is not satisfied, or if the loop is terminated by break statement, the for loop will exit without executing the post statement. For a 3-part for loop, any part of the clause may be empty, but the two semicolons must be present.

In the case where there is only a condition statement in the clause, the two semicolons may be omitted. This style of for loop is called “while loop style” in GoLite. The procedure in executing this is very similar to the while loop in many other languages.

In the case where the condition is also empty, a special kind of infinite loop is defined. Both “for {}” and “for ;; {}” formats are accepted. Of course, infinite loop can also be implemented by using loops with condition that always evaluate to the constant “true”, i.e. “for true{}”. However, this does not fall under infinite loop style according to GoLang’s compiler. Unlike the infinite loops with empty statements, which are considered as terminating if no break statement is found inside their body, the while-loop style infinite loop cannot be used as a terminating statement in neither GoLang or GoLite.

Fortunately, Python does not requires functions to declare their return types in their signatures. Functions in Python also do not have to return a consistent type. Therefore, we will assume that all programs that pass our typechecker have functions with valid terminating statements and we do not have to write any special process for the special infinite loop.

Turning to mapping strategies, Python’s for loop share more similarities to for-range loop in Go, which is not part of GoLite. We will instead use while-loops to implement Go’s for loops. Unlike Go, Python does not have block scoping, so we will simply wrap every for loop inside a function with a name of “___”. Since there is no such a thing as “recursive loop”, it is safe to overshadow or overwrite this variable. In addition, since we will be prefixing “___” to all names, there is no chance this will cause a conflict with anything. The function will be called immediately after its definition and it is then safe to be shadowed by any other generated variables. Since Python’s functions can be defined almost anywhere, this is a straightforward translation.

The for loop’s init statement will go to the first line inside the function, and the condition will be declared as the while loop’s condition, and the post statement will be placed as the last line inside the translated while loop’s body. In addition, the post statement will also be added before any “continue” statements.

3 Summary of Work Done

We have attempted to implement the following: overall structure of code generator, variable declarations with initialization, type declarations, if statements, part of for statements.